Are OVM & UVM Macros Evil? A Cost-Benefit Analysis

Adam Erickson
Mentor Graphics Corporation
890 Winter St.
Waltham, MA 02451
adam_erickson@mentor.com

ABSTRACT

Are macros evil? Well, yes and no. Macros are an unavoidable and integral part of any piece of software, and the Open Verification Methodology (OVM) and Universal Verification Methodology (UVM) libraries are no exception. Macros should be employed sparingly to ease repetitive typing of small bits of code, to hide implementation differences or limitations among the vendors' simulators, or to ensure correct operation of critical features. Although the benefits of the OVM and UVM macros may be obvious and immediate, benchmarks and recurring support issues have exposed their hidden costs. Some macros expand into large blocks of complex code that end up hurting performance and productivity, while others unnecessarily obscure and limit usage of otherwise simple, flexible APIs.[1]

The 'ovm_field macros in particular have long-term costs that far exceed their short-term benefit. While they save you the one-time cost of writing implementations, their run-time performance and debug costs are incurred over and over again. Consider the extent of reuse across thousands of simulation runs, across projects, and, for VIP, across the industry. These costs increase disproportionately with increased reuse, which runs counter to the goals of reuse.

In most cases, it takes a short amount of time and far fewer lines of code to replace a macro with a "direct" implementation. Testbenches would be smaller and run faster with much less code to learn and debug. The costs are fixed and up-front, and the performance and productivity benefits *increase* with reuse.

This paper will:

- Contrast the OVM macros' benefits (what they do for you) with their costs (e.g. inflexibility, low performance, debug difficulty, etc.) using benchmark results and code analysis.
- Identify which macros provide a good cost-benefit trade-off, and which do not.
- Show how to replace high-cost macros with simple SystemVerilog code.
- Provide insight into the work being done to reduce the costs of using macros in the UVM, the OVM-based Accellera standard verification library currently under development.

[1] References to OVM macros shall also apply to UVM macros unless otherwise stated.

1. Introduction

The hidden costs associated with using certain macros may not be discovered until the economies of scale and reuse are expected but not realized. A VIP defined with certain macros incurs more overhead and may become more difficult to integrate in large-scale system-level environments.

The following summarizes our recommendations on each class of macros in the OVM.

Table 1. Summary Macro Usage Recommendations

'ovm_*_utils	*Always use*. These register the object or component with the OVM factory. While not a lot of code, registration can be hard to debug if not done correctly.
'ovm_info | warning | error | fatal	*Always use*. These can significantly improve performance over their function counterparts (e.g. ovm_report_info).
'ovm_*_imp_decl	*OK to use*. These enable a component to implement more than one instance of a TLM interface. Non-macro solutions don't provide significant advantage.
*'ovm_field_**	*Do not use.* These inject lots of complex code that substantially decreases performance, limits flexibility, and hinders debug. Manual implementations are significantly more efficient, flexible, transparent, and debuggable. *In recognition of these faults, the field macros have been substantially improved in the UVM.*
*'ovm_do_**	*Avoid*. These unnecessarily obscure a simple API and are best replaced by a user-defined task, which affords far more flexibility and transparency.
'ovm_sequence-related macros	*Do not use.* These macros build up a list of sequences inside the sequencer class. They also enable automatic starting of sequences, which is almost always the wrong thing to do. *These macros are deprecated in the UVM and thus are not part of the standard.*

Application of these recommendations can have a profound effect. If the 'ovm_field macros were avoided entirely, several thousands of lines of code in the OVM library would not be used, and many thousands more would not be generated (by the macros).

The following section describes the cost-benefit of each macro category in more detail.

2. Cost-Benefit Analyses

2.1 ‘ovm_*_utils

Always use.

The ‘ovm_*_utils macros expand into code that registers the class with the OVM factory, defines the create() method, and, if the type is not a parameterized class, the get_type_name() methods. Because type registration with the factory must be performed in a precise, consistent way, and the code involved is small and relatively straightforward, these macros provide convenience without significant downside.

2.2 ‘ovm_info | warning | error | fatal

Always use.

Issuing a report involves expensive string processing. If the message would be filtered out based on the verbosity, or if it’s configured action is OVM_ACTION, all the string processing overhead would be wasted effort. These report macros improve simulation performance by checking verbosity and action settings before calling the respective ovm_report_* method and incurring the cost of processing the report.

These macros also conveniently provide a report’s location of invocation (file and line number). You can disable file and line number by overriding the ovm_report_server or by defining OVM_REPORT_DISABLE_FILELINE on the command line.

2.3 ‘ovm_*_imp_decl

OK to use.

These macros define special imp ports that allow components to implement more than one instance of a TLM interface. For example, the ovm_analysis_imp calls the host component’s write method, of which there can be only one. Multiple such ovm_analsys_imps would all call the same write method. To get around this, you can invoke the ovm_*_imp_decl macro to define an imp that calls a different method in the component. For example:

```
‘ovm_analysis_imp_decl(_exp)
‘ovm_analysis_imp_decl(_act)
class scorebd extends ovm_component;
  ovm_analysis_imp_exp #(my_tr,scorebd) expect;
  ovm_analysis_imp_act #(my_tr,scorebd) actual;
  virtual function void write_exp(my_tr tr);
    ...
  endfunction
  virtual function void write_act(my_tr tr);
    ...
  endfunction
endclass
```

Writes to the expect_ap analysis imp will call write_expect, and writes to the actual_ap analysis imp will call write_actual.

The imp_decl macros have a narrow use-model, and they expand into a small bits of code. They are OK to use, as they offer a convenience with little downside.

If you do not want to use the *_imp_decl macros, you could implement the following. Define a generic analysis_imp that takes a "policy" class as a type parameter. The imps’ write method calls the static write method in the policy class, which calls a uniquely-named method in the component. You will need to define a separate policy class for each unique instance of the analysis interface, much like what the ovm_*_ imp_decl macros do for you.

```
class aimp #(type T=int, IMP=int, POLICY=int)
    extends ovm_port_base #(tlm_if_base #(T,T));
  `OVM_IMP_COMMON(`TLM_ANALYSIS_MASK,
                  "ovm_analysis_imp",IMP)
  function void write (input T t);
    POLICY::write(m_imp , t);
  endfunction
endclass

class wr_to_A #(type T=int, IMP=int);
  static function void write(T tr, IMP comp);
    comp.write_A(tr);
  endfunction
endclass

class wr_to_B #(type T=int, IMP=int);
  static function void write(T tr, IMP comp);
    comp.write_B(tr);
  endfunction
endclass



class my_comp extends ovm_component;
  aimp #(my_tr, my_comp, wr_to_A) A_ap;
  aimp #(my_tr, my_comp, wr_to_B) B_ap;
  virtual function void write_A(my_tr tr);
   ...
  endfunction
  virtual function void write_B(my_tr tr);
   ...
  endfunction
endclass
```

2.4 ‘ovm_do_*

Avoid.

The ‘ovm_do_* macros comprise a set of 18 macros for executing sequences and sequence items, each doing it a slightly different way. Many such invocations in your sequence body()

method will expand into lots of inline code. The steps performed by the macros are better relegated to a task.

The ‘ovm_do macros also obscure a very simple interface for executing sequences and sequence items. Although 18 in number, they are inflexible and provide a small subset of the possible ways of executing. If none of the ‘ovm_do macro flavors provide the functionality you need, you will need to learn how to execute sequences without the macros. And once you’ve learned that, you might as well code smartly and avoid them all together.

```
virtual task parent_seq::body();
  my_item item;
  my_subseq seq;
  ‘ovm_do(item)  <-- what do these do?
  ‘ovm_do(seq)   <-- side effects? are you sure?
endtask


-----------
task parent_seq::do_item(ovm_sequence_item
item,...);
  start_item(item);
  randomize(item) [with { ... }];
  finish_item(item);
endtask


virtual task parent_seq::body();
 my_item item =
my_item::type_id::create("item",,get_full_name()
);
 my_seq  seq =
my_seq::type_id::create("seq",,get_full_name());
 do_item(item);
 seq.start();
endtask
```

Most uses of the inline constraints seen by this author set the address or data member to some constant. It would be more efficient to simply turn off randomization for those members and set them directly using ’=’. Encapsulating this procedure in a task is also a good idea. A task for simple reads/writes:

```
task parent_seq::do_rw(int addr, int data);
  item= my_item::type_id::create
              ("item",,get_full_name());
  item.addr.rand_mode(0);
  item.data.rand_mode(0);
  item.addr = addr;
  item.data = data;
  item start_item(item);
  randomize(item);
  finish_item(item);
endtask

virtual task parent_seq::body();
  repeat (num_trans)
    do_rw($urandom(),$urandom());
endtask
```

2.5 ‘ovm_sequence macros

Do not use.

The macros, ‘ovm_sequence_utils, ‘ovm_sequencer_utils, ‘ovm_update_sequence_lib[_and_item] macros are used to build up a sequencer’s "sequence library." Using these macros, each sequence type is associated with a particular sequencer type, whose sequence library becomes the list of the sequences that can run on it. Each sequencer also has three built-in sequences: simple, random, and exhaustive.

When a sequencer’s run task starts, it automatically executes the default_sequence, which can be set by the user using set_config. If a default sequence is not specified, the sequencer will execute the built-in ovm_random_sequence, which randomly selects and executes a sequence from the sequence library.

These macros hard-code sequence types to run on a single sequencer type, do not support parameterized sequences, and cause many debug issues related to random execution of sequences. In practice, the sequencer can not start until, say, the DUT is out of reset. When it does start, it typically executes a specific sequence for DUT configuration or initialization, not some random sequence.

Users often spend lots of time trying to figure out what sequences are running and why, and they inevitably look for ways to disable sequence library behavior. (Set the sequencer’s count variable to 0, use ‘ovm_object_utils for sequences, and use ‘ovm_component_utils for sequencers.)

The problems with the sequence library and related macros grow when considering the UVM, which introduces multiple run-time phases that can execute in parallel and in independently timed domains. A single, statically-declared sequence library tied to a single sequencer type cannot accommodate such environments. Therefore, the Accellera VIP-TSC committee decided to officially deprecate the sequence library and macros. The committee is currently developing a replacement sequence library feature that has none of the limitations of its predecessor’s and adds new capabilities.

2.6 ‘ovm_field_*

Avoid.

The ‘ovm_field macros implement the class operations: copy, compare, print, sprint, record, pack, and unpack for the indicated fields. Because fields are specified as a series of consecutive macros calls, the implementation of these operations cannot be done in their like-named do_<operation> methods. Instead, the macros expand into a single block of code contained in an internal method, m_field_automation. Class designers can hand-code field support by overriding the virtual methods— do_copy, do_compare, etc.. Users of the class always call the non-virtual methods—copy, compare, etc.— methods, regardless of whether macros or do_* methods were used to implement them. For example, consider the implementation of the ovm_object::copy non-virtual method:

```
function void ovm_object::copy(...);
```

```
  m_field_automation(COPY,…); //‘ovm_field props
  do_copy(...);           // user customizations
endfunction
```

The non-virtual copy first calls m_field_automation to take care of the ‘ovm_field-declared properties, then calls the corresponding virtual do_ copy to take care of the hand-coded portion of the implementation.

Because of the way the ‘ovm_field macros are implemented and the heavy use of policy classes (comparer, printer, recorder, etc.), macro-based implementations of the class operations incur high overhead. The next few sections provide details on this and other costs..

2.6.1 Code bloat

Consider the simple UBUS transaction definition below.[2]

```
class ubus_transfer extends ovm_sequence_item;

  rand bit [15:0]           addr;
  rand ubus_op              op;
  rand int unsigned         size;
  rand bit [7:0]            data[];
  rand bit [3:0]            wait_state[];
  rand int unsigned         error_pos;
  rand int unsigned         transmit_delay = 0;
  string                    master = "";
  string                    slave = "";

  `ovm_object_utils_begin(ubus_transfer)
    `ovm_field_int  (addr,           UVM_ALL_ON)
    `ovm_field_enum (ubus_op, op,    UVM_ALL_ON)
    `ovm_field_int  (size,           UVM_ALL_ON)
    `ovm_field_array_int(data,       UVM_ALL_ON)
    `ovm_field_array_int(wait_state, UVM_ALL_ON)
    `ovm_field_int  (error_pos,      UVM_ALL_ON)
    `ovm_field_int  (transmit_delay, UVM_ALL_ON)
    `ovm_field_string(master,UVM_ALL_ON |
                             UVM_NOCOMPARE)
    `ovm_field_string(slave,UVM_ALL_ON |
                             UVM_NOCOMPARE)
  `ovm_object_utils_end

endclass
```

After macro expansion, this 22-line transaction definition expands to 644 lines, a nearly 30-fold increase. Real-world transaction definitions far exceed 1,000 lines of code. The following table shows the number of new lines of code that each of the ‘ovm_field macros expand into, for both OVM 2.1.1 and UVM 1.0. In UVM 1.0, the macros underwent significant refactoring to improvement performance and provide easier means of manually implementing the do_* methods.

Table 1 Macro expansion – lines of code per macro

Macro	*Lines of Code OVM*[3]	*Lines of Code UVM*[2]
`ovm_field_int|object|string|enum	*51,72,17,41*	*50,75,43,45*
*‘ovm_field_sarray_**	*75-100*	*117-128*
*‘ovm_field_array_**	*127-191*	*131-150*
*‘ovm_field_queue_**	*110-187*	*133-152*
‘ovm_field_aa_*_string	*76-87*	*75-102*
‘ovm_field_aa_object_int	*97*	*111*
*‘ovm_field_aa_int_**	*85*	*85*
‘ovm_field_event	*16*	*29*

In contrast, the manual implementation of the same UBUS transaction consists of 92 lines of code that is more efficient and human-readable.

2.6.2 Low performance

The lines of code produced by the expansion of the ‘ovm_field macros do not actually do much of the actual work. That is handled by nested calls to internal functions and policy classes (e.g. ovm_comparer, ovm_printer, etc.).

Table 2 shows how many function calls are made by each operation for the macro-based solution and the equivalent manual implementation of the do_ methods. As a control, the size of the `data` and `wait_state` members were fixed at 4.

Table 2 Function calls per UBUS operation

Operation	*OVM Macro/Manual*	*UVM Macro/Manual*
copy	*38 / 9*	*8 / 9*
compare	*51 / 18*	*17 / 18*
sprint - table	*1957 / 1840*	*187 / 160*
sprint - tree	*518 / 441*	*184 / 157*
sprint – line	*478 / 405*	*184 / 157*
pack / unpack	*140 / 28*	*80 / 28*
record (begin_tr / end_tr)	*328 / 46*	*282 / 36*

Compare these results with a theoretical minimum of one or two calls, depending on whether the object has a base class. Calling copy in a macro-based implementation incurs 38 function calls, but only 9 in a do_compare implementation—a four-fold difference. Compare incurs 51 method calls with macros versus do_compare’s 18 calls. Sprinting (and printing) incur thousands of calls for each operation.

Each function call involves argument allocation, copy, and destruction, which affects overall performance. The results were alarming enough that significant effort was taken to improve the macro implementations in UVM. The UVM column shows this.

[2] The UBUS is a contrived bus protocol used in examples in the UVM 1.0 User Guide. It’s predecessor in OVM was XBUS.

[3] ‘ovm_field_aa_* macros do not implement record, pack, or unpack; line counts would be much greater if they did.

Table 3 shows the run time to complete 500K operations for the macro-based and manual implementations of the do_* methods.

Table 3 Performance – 500K transactions, in *seconds*[4]

Operation	*OVM Macro/Manual*	*UVM Macro/Manual*
copy	*43 / 2*	*8 / 2*
compare	*60 / 6*	*9 / 6*
sprint - table	*1345 / 1335*	*165 / 159*
sprint - tree	*215 / 165*	*137 / 137*
sprint – line	*195 / 165*	*137 / 132*
pack / unpack	*100 / 19*	*37 / 18*
record (begin_tr/end_tr)	*533 / 40*	*413 / 37*

The poor performance results in OVM prompted a significant effort to improve them in UVM. The results of this improvement effort show that performance issues for most operations have largely been mitigated.

Amdahl's Law [5] states that testbench performance improvements are limited by those portions of the testbench that cannot be improved. Although this author still cannot recommend field macro usage over manual implementation, the macro performance improvements in UVM are very welcome because they afford significant performance improvements achievable in emulation and acceleration.

Note that the sprint times are comparable between the macro-based and manual implementations. This is because there is no equivalent manual replacement for the formatting capabilities of the printer policy class, the primary source of overhead for this method. The UVM provides an improved uvm_printer policy class that makes performance less sensitive to output format.

2.6.3 Not all types supported

The 'ovm_field macros do not support all the type combinations you may need in your class definitions. The following are some of the types that do *not* have 'ovm_field macro support.

- Objects not derived from ovm_object
- Structs and unions
- Arrays (any kind) of events
- Assoc arrays of enums
- Assoc arrays of objects indexed by integrals > 64 bits
- Assoc arrays—no support for pack, unpack, and record
- Multi-dimensional packed bit vectors—For example, bit [1:3][4:6] a[2]. The [1:3][4:6] dimensions will be flattened, i.e. treated as a single bit vector, when printing and recording.
- Multi-dimensional unpacked bit vectors— For example, bit a[2][4]
- Multi-dimensional dynamic arrays, such as arrays of arrays, associative array of queues, etc.

2.6.4 Debugging difficulties

The 'ovm_field (and, still, the `uvm_field) macros expand into many lines of complex, uncommented code and many calls to internal and policy-class methods.

If a scoreboard reports a miscompare, or the transcript results don't look quite right, or the packed transaction appears corrupted, how is this debugged? Macros would have been expanded, and extra time would be spent stepping through machine generated code which was not meant to be human readable.

The person debugging the code may not have had anything to do with the transaction definition. A single debug session traced to the misapplication, limitation, or undesirable side effect of an `ovm_field macro invocation could negate the initial ease-of-implementation benefit it was supposed to provide. Manually implementing the field operations *once* will produce more efficient, straight-forward transaction definitions.

As an exercise, have your compiler write out your component and transaction definitions with all the macros expanded.[5] Then, contrast the macro-based implementations with code that uses straight-forward SystemVerilog:

```
function bit my_obj::do_compare(ovm_object rhs,
                                uvm_comparer comparer);
  do_compare =
    ($cast(rhs_,rhs) &&
     super.do_compare(rhs,comparer) &&
     cmd  == rhs_cmd &&
     addr == rhs_.addr &&
     data == rhs_.data);
endfunction
```

2.6.5 Other limitations

The 'ovm_field macros have other limitations:

- Integrals variables cannot exceed 'OVM_MAX_STREAMBITS bits in size (default is 4096). Changing this global max affects efficiency for all types.
- Integrals are recorded as 1K bit vectors, regardless of size. Variables larger than 1K bits are truncated.
- The ovm_comparer is not used for any types other than scalar integrals, reals, and arrays of objects. Strings, enums, and arrays of integral, enum, and string types do not use the ovm_comparer. Thus, if you were to define and apply a custom comparer policy, your customizations

[4] Simulation results depend on many factors: simulator, CPU, memory, OS, network traffic, etc. Individual results will differ, but relative performance should be consistent.

[5] For Questa, the vlog option is -Epretty <filename>.

- The ovm_packer limits the aggregate size of all packed fields to not exceed OVM_MAX_PACKED_BITS. This large, internal bit vector is bit-by-bit copied and iterated over several times during the course of the pack and unpack operations. If you need to increase the max vector size to avoid truncation, you will affect efficiency for all types.

2.6.6 Dead code

The 'ovm_field macros' primary purpose is to implement copy, compare, print, record, pack, and unpack for *transient* objects. None of these operations are particularly useful to OVM components. Components cannot be copied or compared, and pack and unpack doesn't apply. Print for components are occasionally useful for debugging component topology at start of simulation, but you could get that and more from a GUI debugger without having to modify the source. In most cases, a simple $display("%p",component) would suffice.

The 'ovm_field macros also implement a little-known feature called auto-configuration, which performs an implicit get_config for every property you declare with an 'ovm_field macro inside an ovm_component. While convenient sometimes, it presumes all macro-declared fields are intended to be user-configurable, and you sacrifice control over whether, when, and how often configuration is retrieved. For ovm_objects, auto-config code is never used. For ovm_components, this feature incurs significant time to complete and is in many cases unwanted. To avoid this overhead, users often disable auto-config by not calling super.build() and simply call get_config explicitly for the properties intended to be user-configurable.

Despite performance improvements in UVM, the field macros still incur code bloat, performance degradation, debug issues, and other limitations. The UVM also provides small convenience macros for helping users manually implement the do_* methods more easily. For these reasons, this author continues to recommend against using the field macros.

3. Alternative to 'ovm_field macros

The following sections describe how to write implementations of copy, compare, etc. without resorting to the 'ovm_field macros. In all cases, you override the do_<method> counterpart. For example, to manually implement copy, you override the virtual do_copy method. For UVM, change the O's to U's.

3.1 do_copy

Implement the `do_copy` method as follows:

```
1  function void do_copy (ovm_object rhs);
2    my_type rhs_;
3    if (!$cast(rhs_,rhs))
4      `ovm_fatal("TypeMismatch","...");
5    super.do_copy(rhs);
6    addr = rhs_.addr;
7    if (obj == null && rhs_.obj != null)
8      obj = new(...);
9    if (obj!=null) obj.copy(rhs_.obj);
10 endfunction
```

Line 1—This is the signature of the do_copy method inherited from ovm_object. Your signature must be identical.

Lines 2-4— Copy only works between two objects of the same type. These lines check that the rhs argument is the same type. If not, a FATAL report is issued and simulation will exit.

Line 5—Here, we call do_copy in the super class so any inherited data members are copied. If you omit this statement, the rhs object will not be fully copied.

Line 6—Use the built-in assignment operator (=) to copy each of the built-in data types. For user-defined objects, assignment is copy-by-reference, which means only the handle value is copied. This leaves this object and the rhs object pointing to the same underlying object instance.

Lines 7-9—To deep copy the rhs object's contents into this object, call its copy method. Make sure the obj handle is non-null before attempting this.

3.2 do_compare

Implement the `do_compare` method as follows:

```
1 function bit do_compare (ovm_object rhs,
                           ovm_comparer comparer);
2   my bus op manual rhs ;
3   do_compare =
4     ($cast(rhs_,rhs) &&
5      super.do_compare(rhs,comparer) &&
6      addr == rhs_.addr &&
7      obj != null && obj.compare(rhs_.obj)
9     );
10 endfunction
```

Line 1—This is the signature of the do_compare method inherited from ovm_object. Your signature must be identical.

Line 3—This line begins a series of equality expressions logically ANDed together. Only if all terms evaluate to true will do_compare return 1. Should any term fail to compare, there is no need to evaluate subsequent terms, as it will have no effect on the result. This is referred to as short-circuiting, which provides an efficient means of comparing. We don't need to check the rhs object for null because that is already done before do_compare is called. Be sure to use triple-equal (===) when comparing 4-state (logic) properties, else x's will be treated as "don't care."

Lines 4-— Compare only works between two objects of the same type. The $cast evaluates to 'true' if the cast succeeds, thereby allowing evaluation of subsequent terms in the expression. If the cast fails, the two objects being compared are not of the same type and comparison fails early.

Line 5—Here, we call do_compare in the super class so any inherited data members are compared. If you omit this expression, the rhs object will not be fully compared.

Lines 6—The equality operator (==) can be used to compare any data type. For objects, it compares only the reference handles, i.e. it returns true if both handles point to the same underlying object. You should have one of these expressions for each member you wish to compare.

Lines 7-8—To compare different instances of a class type, call the object's compare method. Make sure the object handle is non-null before attempting this.

3.3 convert2string

The convert2string method is used to print information about an object in free-format. It is as efficient and succinct as the class designer wants, imposing no requirements on the content and format of the string that is returned. The author recommends implementing convert2string for use in `uvm_info messages, where users expect succinct output of the most relevant information.

```
1 function string convert2string();
2   return $sformatf("%s a=%0h, s=%s,
                      arr=%p obj=%s ",
       super.convert2string(), // base class
       addr,                   // integrals
       str,                    // strings
       arr,                    // unpacked types
       obj.convert2string());// objects
3 endfunction
```

Line 1—This is the signature of the convert2string method inherited from ovm_object. Your signature must be identical.

Line 2—This line returns a string that represents the contents of the object. Note that it leverages the built-in $sformatf system function to perform the formatting for you. Use format specifiers to %h, %d, %b, %s, etc. to display output in hex, decimal, binary, or string formats. For unpacked data types, like arrays and structs, use %p for the most succinct implementation. Be sure to call super.convert2string.

3.4 do_print

To implement both print and sprint functionality, you only need to override do_print as follows:

```
1 function void do_print (ovm_printer printer);
2   super.do_print(printer);
3   printer.print_generic("cmd","cmd_t",
                          1,cmd.name());
4   printer.print_field("addr",addr,32);
5   printer.print_array_header("data",
                               data.size(),
                               "byte[$]");
6   foreach(data[i])
7    printer.print_generic($sformatf("[%0d]",i),
                           "byte",
                           8,
                          $sformatf("%0h",data[i]));
8  printer.print_array_footer(data.size());

9 endfunction
```

Line 1—This is the signature of the do_print method inherited from ovm_object. Your signature must be identical.

Line 2—Call super.do_print() to print the base class fields.

Line 3-4—We call methods in the ovm_printer class that correspond to the type we want to print. Enum types use the print_generic method, which has arguments for directly providing field name, type, size, and value.

Line 5-8—Print arrays by printing its header, elements, and footer in separate statements. To print individual elements, the author recommends using print_generic, which allows you to customize what is printed for the element name, type name, and value.

3.5 do_record

Implement do_record as follows. First, define a simple macro, 'ovm_record_field, that calls the vendor-specific system function for recording a name/value pair, e.g. $add_attribute. The macro allows you to pass the actual variable—not some arbitrarily large bit-vector—to $add_attribute. (The UVM will provide these macro definitions for you.)

```
'ifdef QUESTA
   `define ovm_record_att(HANDLE,NAME,VALUE) \
        $add_attribute(HANDLE,VALUE,NAME);
'endif
'ifdef IUS
   'define ovm_record_att(HANDLE,NAME,VALUE) \
         <Cadence Incisive implementation>
'endif
'ifdef VCS
   'define ovm_record_att(HANDLE,NAME,VALUE) \
         <Synopsys VCS implementation>
'endif
`define ovm_record_field(NAME, VALUE) \
    if (recorder != null &&
        recorder.tr_handle!=0) begin \
          `ovm_record_att(recorder.tr_handle, \
                          NAME,VALUE) \
  end
```

These macros serve as a vendor-independent API for recording fields from within the do_record method implementation. Note that, for these macros to work, the ovm_recorder::tr_handle must be set via a previous call to ovm_component::begin_tr or ovm_transaction::begin_tr.

The do_record method simply invokes the `uvm_record_field macro for each of the fields you want recorded:

```
1 function void do_record(ovm_recorder recorder);
2   super.do_record(recorder);
3   `ovm_record_field("cmd",cmd.name()) // enum
4   `ovm_record_field("addr",addr) // integral
5   foreach (data[index])          // arrays
6     `ovm_record_field(
      $sformatf("data[%0d]",index), data[index])
7   obj.record(recorder);          // objects
endfunction
```

Line 1—This is the signature of the do_record method inherited from ovm_object. Your signature must be identical.

Line 2—Be sure to call super.do_record so any inherited data members are recorded.

Lines 3-7—Records enums, integral types, arrays, and objects using invocations of the ‘ovm_record_field macro, or calling a sub-object’s record method.

3.6 do_pack / do_unpack

These operations must be implemented such that unpacking is the exact reverse of packing. Packing an object into bits then unpacking those bits into a second object should be equivalent to copying the first object into the second.

Packing and unpacking require precise concatenation of property values into a bit vector, else the transfer would corrupt the source object’s contents.

To help reduce coding errors, the author advises using small convenience macros.[6] These types of macros are “less evil” because they expand into small bits of readable code that users might otherwise have to write themselves. In fact, the UVM will offer versions of these macros to facilitate robust manual implementations of do_pack and do_unpack.

```
`define ovm_pack_intN(VAR,SIZE) \
   packer.m_bits[packer.count +: SIZE] = VAR; \
   packer.count += SIZE;

`define ovm_pack_array(VAR,SIZE) \
    `ovm_pack_scalar(VAR.size(),32) \
    foreach (VAR `` [index]) begin \
      packer.m_bits[packer.count+:SIZE]=\
                           VAR[index]; \
      packer.count += SIZE; \
    end

`define ovm_pack_queueN(VAR,SIZE) \
   `ovm_pack_arrayN(VAR,SIZE)

`define ovm_unpack_intN(VAR,SIZE) \
   VAR = packer.m_bits[packer.count +: SIZE]; \
   packer.count += SIZE;

`define ovm_unpack_enumN(TYPE,VAR,SIZE) \
   VAR = TYPE'(packer.m_bits[packer.count +: \
                                    SIZE]); \
   packer.count += SIZE;

`define ovm_unpack_queueN(VAR,SIZE) \
    int sz; \
    `ovm_unpack_scalar(sz,32) \
    while (VAR.size() > sz) \
      void'(VAR.pop_back()); \
    for (int i=0; i<sz; i++) begin \
      VAR[i]=packer.m_bits[packer.count+:SIZE];\
      packer.count += SIZE; \
    end

`define ovm_pack_int(VAR) \
    `ovm_pack_intN(VAR,$bits(VAR))
`define ovm_unpack_enum(VAR,TYPE) \
    `ovm_unpack_enumN(VAR,$bits(VAR),TYPE)
`define ovm_pack_queue(VAR) \
    `ovm_pack_queueN(BAR,$bits(VAR[0])
```

The ‘ovm_pack_int macro works for scalar built-in integral types. You can add your own simple macros to support other types, if you like. For example, reals would need the $realtobits and $bitstoreal system functions.

The macro implementations manipulate the m_bits and count properties of the packer object. m_bits is the bit vector that holds the packed object, and count holds the index at which the next property will be written to or extracted from m_bits.

With these *simple* macros defined, you can implement pack and unpack as follows:

```
1 function void do_pack(ovm_packer packer);
2    super.do_pack(packer);
3    `ovm_pack_int(cmd)
4    `ovm_pack_int(addr)
5    `ovm_pack_queue(data)
6  endfunction
7
8  function void do_unpack (ovm_packer packer);
9    super.do_unpack(packer);
10   `ovm_unpack_enum(cmd_t,cmd)
11   `ovm_unpack_int(addr)
12   `ovm_unpack_queue(data)
13 endfunction
```

Line 1—This is the signature of the do_pack method inherited from ovm_object. Your signature must be identical.

Line 2—Always call super.do_pack first.

Lines 3-5—For each property, invoke one of the convenience macros, which concatenates values into the packer’s internal m_bits field and updates the count variable. Here, we’ve leveraged some convenience macros to make it simple and less error prone.

Line 8—This is the signature of the do_unpack method inherited from ovm_object. Your signature must be identical.

Line 9—Always call super.do_unpack first.

Lines 10-12—You *must* unpack each property in the same order as you packed them. You will need to cast the bits when unpacking into strongly typed data types like strings and enums.

4. Conclusion

This paper has provided insight into the hidden costs behind the various macros provided in OVM. Some macros expand into small bits of code that the user would end up writing, or ensure the correct operation of critical features in the OVM. Other macros expand into large blocks of unreadable code that end up hurting performance and productivity in the long run, or unnecessarily obscure and limit usage of a simple, flexible API.

In summary:

We recommend always using the ‘ovm_*_utils macros and the reporting macros: ‘ovm_info, ‘ovm_warning, ‘ovm_warning, and ‘ovm_fatal. These macros provide benefits that far exceed their costs.

The ‘ovm_*_imp_decl macros are acceptable because they provide a reasonable trade-off between convenience and complexity.

[6] Simulators supporting bitstream operators should make packing and unpacking easier, less error prone, and macro free:

bits = {<<{ cmd, addr, data.size(), data, …};

The ‘ovm_field macros have long-term costs that far exceed their short-term benefit. They save you the one-time cost of writing implementations. However, the performance and debug costs are incurred over and over again. Consider the extent of reuse across thousands of simulation runs, and across projects. For VIP, reuse extends across the industry. The more an object definition is used, the more costly ‘ovm_field macros become in the long-run. While the UVM improves the performance of the field macros, it also provides “less evil” macros that help make the do_* methods easier to implement. In this author’s opinion, it is still better to implement simple, manual implementations.

The ‘ovm_do macros attempt to hide the start, start_item, and finish_item methods for sequence and sequence_item execution. This is unnecessary and confusing. The current 18 macro variants with long names and embedded in-line constraints cover only small fraction of the possible ways you can execute sequences and sequence items. It is easier to learn to use the simple 3-method API, encapsulating repeated operations inside a task.

The ‘ovm_sequence-related macros hard-code a sequence to a particular sequencer type and facilitate the auto-execution of random sequences. Sequences should not be closely couple to a particular sequencer type, and they should not be started randomly. Stimulus generation is typically preceded by reset and other initialization procedures that preclude their automatic execution. You should declare sequences with ‘ovm_object_utils and sequencers with ‘ovm_component_utils, then start specific sequences explicitly using the start method. The UVM recognizes these and other shortcomings by deprecating the macros and OVM sequence library API. A new, superior sequence library implementation that is decoupled from the sequencer is currently being developed.

5. Acknowledgements

The author wishes to acknowledge the significant collaborative contributions of John Rose, Senior Product Engineer at Cadence Design Systems, Inc., toward improving the performance of the field-macro definitions in the UVM.

6. References

[1] "IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009.

[2] OVM 2.1.1 Reference, ovmworld.org

[3] OVM User Manual, ovmworld.org

[4] Accellera Verfication IP Technical SubCommittee (UVM Development Website); http://www.accellera.org/apps/org/workgroup/vip

[5] Amdahl’s Law: http://en.wikipedia.org/wiki/Amdahl’s_law